# An experience report on the integration of ECU software using an HSF-enabled real-time kernel*

Martijn M.H.P. van den Heuvel, Erik J. Luit, Reinder J. Bril,
Johan J. Lukkien, Richard Verhoeven and Mike Holenderski

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven (TU/e),
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

***Abstract*—This paper gives an overview of the challenges we faced when integrating automotive software components on an embedded electronic control unit (ECU). The results include the design of a communication abstraction layer, management of scarce ECU resources and a demonstration of temporal isolation between components in an industrial case study.**

***Index Terms*—Automotive software; Virtualization; Real-time scheduling; Component-Based Software Engineering (CBSE).**

## I. Introduction

Today's vehicles contain an ever increasing amount of software. These software functions consist of various components that replace mechanical controllers. The current market situation reinforces the challenges of integrating these software functions on a shared platform, because adding a new function into a vehicle often means purchasing pre-manufactured hardware and software with little information about the internal behavior [1].

The AUTOSAR consortium, however, recognized that a revolutionary performance increase of in-vehicle electronic systems comes from the composition and the integration of independently developed software functions. In AUTOSAR, functions are developed using components which are executed as tasks by an OSEK-certified operating system (OS). Some of these tasks may share memory-mapped input-and-output (I/O) devices, actuation devices (such as brakes) and software pieces [1] (such as object detection). The protocols that manage synchronization on these shared resources may further impact I/O delays experienced by the tasks of a component. Many components, especially those that implement control functionality, are sensitive to timing and fluctuations in actuation delays.

Hierarchical scheduling frameworks (HSFs) support promising techniques to control such timing delays and fluctuations. In order to support composition of components and temporal isolation between them, Nolte et al. [2] investigated the applicability of HSFs into AUTOSAR. The HSF is implemented using so-called *servers* as a layer between the AUTOSAR OS and the AUTOSAR Runtime Environment. The AUTOSAR standard allows for inclusion of proprietary technology, as long as the extensions can be abstracted to an AUTOSAR OS [2]. In this work we apply an HSF to real automotive software and we demonstrate its use in the field by means of video material.

*This work is supported by the Dutch High-Tech-Automotive-Systems innovation programme under the VERIFIED project (Grant number: HTASI10003).

The remainder of this paper is organized as follows. Section II gives a brief overview of the case study being explored in this paper. Section III then presents the software components that were developed for our use case. Section IV describes the deployment of those software modules on our ECU. Section V discusses some of the practical challenges we faced in the development and deployment of our ECU software. Finally, Section VI concludes this paper.

## II. An automotive case study

In this work we integrated 3 software applications into a Jaguar XF (see Figure 1): an active suspension controller [3], a supervisory controller and a run-away process. We established timing predictable execution of these applications by means of an HSF, which allocates a server to each application.

The active suspension is part of a more comprehensive Integrated Vehicle Dynamics Controller (IVDC), which is meant to stabilize a vehicle in critical situations. The IVDC further improves the electronic stability program (ESP) of a car by adding suspension control to the integrated control [3].

A supervisory controller checks the correctness of the shared sensor and actuator data and handles faults when necessary. It is split up in a Central Supervisory Control (CSC) which coordinates central actions for the 4 wheels and a Local Supervisory Control (LSC) which controls a single suspension unit for one wheel. More precisely, the CSC implements logic to coordinate the suspension per axle and for the entire car.

The run-away process can be put in a mode where it consumes all processor cycles and it runs at the highest priority. It is used to demonstrate temporal isolation between the three applications, i.e., each application can consume only the resources allocated to its server and nothing more.

### A. Logical view to hardware

We use various ECUs in the car which are connected to a fieldbus; some of these nodes are virtual ones. Each wheel is controlled locally. In our setup, one wheel is controlled by an ECU while the other wheels are controlled by a dSPACE [4] system (hence, the other ECUs are not deployed in real and their software runs on a central dSpace node). dSPACE provides a powerful hardware platform and tools for prototyping embedded applications. The CSC also executes on the dSPACE system.

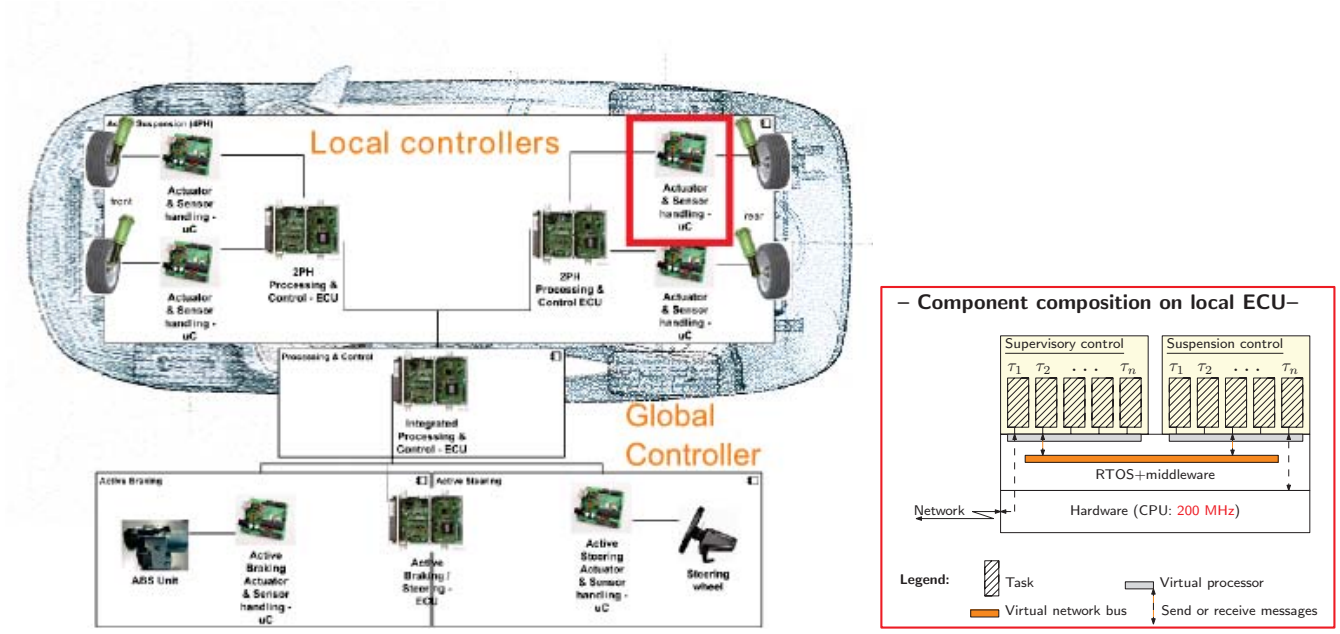


Figure 1. In this project we employed 4 ECUs at each corner of a test car for active suspension. A central dSpace coordinates the local ECUs. It therefore implements components for supervisory control and software-based integrated-vehicle-dynamics (IVDC) state estimation. We have integrated their local counterparts, i.e., 2 components which are (semi-)independently developed by various project partners, through an HSF with well-defined mechanisms for resource virtualization on a local ECU.

The ECU that we used is a Freescale EVB9S12XF512E evaluation board with a 16-bits, MC9S12XF512 processor and 32 kB on-chip RAM. The clock speed of the processor was set to 40MHz in order to accommodate the processing load. The board provides, among others, 16 Analog to Digital Converters (ADCs), several PWM outputs, a CAN controller and a FlexRay controller. The Freescale board is connected to an extension board which protects the processor hardware from electric overloads, it offers voltage division and it provides connectors to the processor board and to the environment.

### B. This work

In this work, a dedicated ECU is deployed in order to control the suspension of one of the four wheels of a car. On this ECU, we implement and run three different applications:

- **Two control loops for active suspension:** these tasks run at 400 Hz and 100 Hz, respectively (i.e., tasks with periods of 2.5 ms and 10 ms). These loops execute a control model (developed using Matlab/Simulink) and they interact directly with the hardware.
- **The LSC process:** it receives commands from the CSC, sends commands to the control loops, receives data from the control loops and sends state information to the CSC.
- **Run Away Process (RAP):** on command it switches between a state in which it sends an *"I'm alive"* message each period and a state in which it tries to consume all CPU cycles.

Using our HSF extensions in MicroC/OS-II, temporal isolation is demonstrated between the three applications. Hence, the other applications are protected against the RAP. Moreover, we describe their mapping on a platform with scarce resources.

## III. ECU Software

In this section we give an overview of the software modules that are integrated on our ECU. Figure 2 shows the

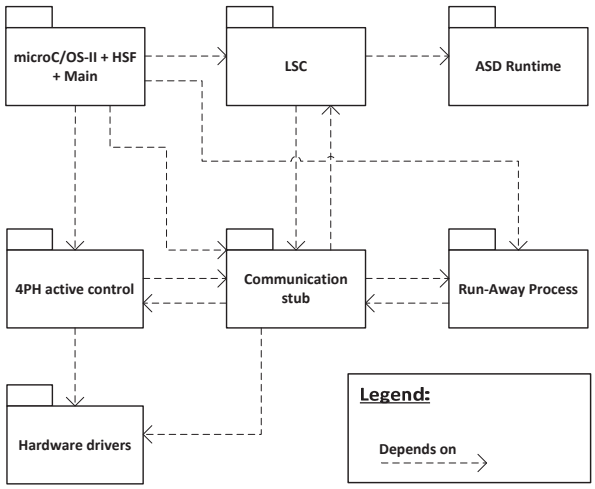


Figure 2. An overview of the software modules, including their dependencies, which we have integrated in our ECU.

dependencies between the different modules. Firstly, we briefly recapitulate MicroC/OS-II and its HSF. Secondly, we introduce the LSC and its run-time libraries. Thirdly, we introduce the 4-point-hydraulic (4PH) suspension control. Finally, we describe our communication stub.

The hardware drivers are not further described. These drivers were mostly delivered with MicroC/OS-II or by Freescale. Moreover, the most interesting part is described by our communication stub[1], which provides an abstraction layer for the underlying fieldbus drivers (either CAN or FlexRay).

Also the RAP is not discussed in further detail. The reason is that the RAP is a fairly simple process, i.e., an event-triggered infinite loop which is introduced for the purpose of demonstrating temporal isolation within an HSF.

[1]The dependencies of the communication stub to the application modules (LSC, RAP and 4PH active control) are just there to ease their definitions of message types; they can be avoided by means of singleton-like patterns.

### A. MicroC/OS-II and its HSF

MicroC/OS-II is a microkernel which is maintained and supported by Micrium [5] and is applied in many application domains, e.g., automotive[2]. The kernel is open source and available for free for non-commercial purposes. The MicroC/OS-II kernel features preemptive multitasking for up to 256 tasks, and its size is configurable at compile time, e.g. services like mailboxes and semaphores can be disabled.

This section recapitulates our proprietary HSF module for MicroC/OS-II [6, 7]. Extending MicroC/OS-II with basic HSF support requires a realization of the following concepts:

1) *Server scheduling:* Similar to the MicroC/OS-II task scheduling approach, we introduce a ready queue for servers indicating whether or not a server has capacity left. When the scheduler is called, it activates the *ready* server with the highest priority. The fixed-priority scheduler of MicroC/OS-II then selects the highest-priority ready task from the group of tasks corresponding to the running server. The implementation of periodic servers turned out to be very similar to implementing periodic tasks [6].
2) *Task scheduling:* After masking the task groups of all servers except the tasks of the active one, the MicroC/OS-II fixed-priority scheduler subsequently determines the highest priority ready task; this code is unmodified.
3) *Idle Server:* We reserve the lowest task priority levels for an idle server, which contains MicroC/OS-II's idle task at the lowest local priority. This server cannot deplete its budget, so that the idle server can always be switched in whenever no other server is eligible to execute.

A major effort in the HSF's realization translates into a hierarchical representation of timed events. In a system we therefore employ four timer queues to control tasks and servers. In case of single level scheduling, we have just a single system queue that represents the timer events associated with the arrival of tasks. In an HSF, we use this existing system queue for the scheduling of servers. The timers in this queue represent budget-replenishment events corresponding to the start of a new period. In addition there is a local queue for each server which keeps track of the timers needed to manage the tasks inside a server such as the arrival of periodic tasks. At any time at most one server can be running on the processor; all other servers are inactive. When a server is suspended, its local queue is deactivated. In this configuration the hardware timer drives two timer queues, i.e., the local queue of the active (running) server and a system queue.

When the running server is preempted, its local queue is deactivated and the queue belonging to the newly scheduled server is activated. In order to ensure correct execution, the time that passed since the previous deactivation needs to be accounted for upon activation. To keep track of this time we introduce a third queue: the *stopwatch queue*. Upon deactivation of a server, a timer is added to this queue. Whenever a server is activated, its local queue is synchronized with the stopwatch, i.e., all timers in its local queue which would have expired if the server was running are handled. As a result, all local timers with a smaller value than the stopwatch timer are popped from the local queue and the corresponding stopwatch event is subsequently deleted from the stopwatch queue. The time spent to synchronize the local queue of the newly activated server with global time is accounted to this server and subtracted from its budget.

Finally, a fourth queue represents timers that expire relative to the server budget. These events trigger the depletion of (a fraction of) the server's budget. We call these *virtual timers* as their notion of time is limited to the server budget. Rather than putting these in the system queue we have a separate queue for them, since otherwise we would need to insert them into the system queue upon activation and remove them again upon deactivation. In this new configuration, at every tick interrupt at most four queues are updated: a system queue, an active server queue, a stopwatch queue, and an active server virtual queue. The last queue does not need to get synchronized when a server is resumed, because a deactivated server does not consume its budget.

We refer the interested reader to [6] for a detailed performance evaluation of MicroC/OS-II and our HSF.

### B. Local supervisory control and its ASD runtime

The Local Supervisory Control (LSC) consists of code generated from formally verified state charts. These state charts are programmed using the *ASD:Suite* [12]. Although ASD's underlying model-checking techniques can guarantee absence of faults in the state-chart models, absence of faults is not automatically guaranteed in the modeled program unless code generation techniques are applied.

For this purpose, amongst other approaches, Broadfoot and Broadfoot [8] proposed to bridge the gap between formal methods and the informal world of software engineering by combining the sequence-based specification method (SBS) [9] and the process algebra Communicating Sequential Processes (CSP) [10]. Broadfoot and Hopcroft [8, 11] extended this work by developing automated translations between SBSs, CSP and executable code, such that the operational semantics are preserved. This led to the invention of Analytical Software Design (ASD) and together with the commercial product ASD:Suite [12], developed and owned by Verum, enables its full integration into industrial practices.

Using ASD, we describe the provided interface of the LSC component, which consists of the following methods:

- *comm_ok*;
- *controls_enabled*;
- *reset_system*;
- *reset_errors*.

These methods can be called by other components in the system, i.e, in our case, the communication stub.

The behaviour behind the interface of the LSC component is then captured by a state chart, as shown in Figure 3. It has the following states: uninitialized, passive and active. The state changes of the LSC are triggered by the received commands

[2]Unfortunately, the suppliers of MicroC/OS-II have discontinued the support for an OSEK-compatibility layer.

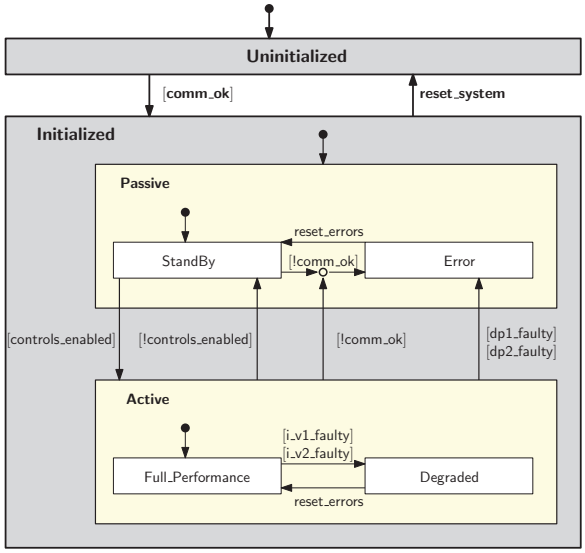


Figure 3. A state-chart representation of the LSC.

from incoming network messages. From the uninitialized state, a transition is made to the initialized/standby state when the LSC receives the *comm_ok* message from the communication stub. This message is sent as soon as the first message is received from the CSC. When the *controls_enabled* signal is received, the active/full performance state is entered.

When faults are detected, the LSC goes into either the degraded state or into the passive state. The degraded state is entered if the measured sensor data (i.e, the pressure and current) deviate from their expected values. These errors are reported to the CSC and the LSC can return to the full performance state when the *reset_errors* message is received.

In the passive states the local 4PH suspension control acts independently of the central control. This happens, e.g., when the communication between the dSpace box and our ECU fails. The communication is considered to be correct (see Section V-B) as long as maximally two messages from the CSC are missed, either because these did not arrive at the ECU or because the ECU could not process these in time. If the communication fails, i.e., when more than two messages are not received, then the passive state is entered. When messages are arriving again, the communication stub sends the *comm_ok* message again and the full performance state is re-entered. Otherwise, the LSC stays in the passive state.

Finally, the above design is formally verified by ASD. By modeling its environment, e.g., the interface of the communication stub which may use the LSC's provided interface, concurrency issues of tasks interacting with the LSC can be avoided. Subsequently, MISRA C compliant source code [13] has been generated which implements the model.

### C. 4PH active suspension

The local 4PH suspension control of our ECU controls the suspension unit at one wheel of the car. The suspension unit for one wheel consists of a conventional suspension extended with a hydraulic system. The hydraulic system consists of a fluid-filled cylinder with a piston that divides the cylinder into two parts. The pressure on both sides of the piston can be varied by two electrically operated valves, so that the piston and the rod attached to it can move in both directions. The valves are actuated via Pulse Width Modulation (PWM), so the effective voltage applied is determined by the ratio of the duty cycle and the period of the PWM. Hydraulic pressure is measured on both sides of the valves. Also the actual current of the valves and the voltage of the power supply are measured.

The code generated from this active-suspension application for our ECU consists of 2 control loops: one controls the pressure of the valves at 100Hz and the other controls the current at 400Hz. The central dSpace box runs the software for the other 3 wheels of the car and it runs the CSC which implements logic to coordinate the suspension per axle and for the entire car. The entire control application has been modelled and tested using Matlab-Simulink. For details on the vehicle dynamics, we refer the interested reader to [3].

### D. Communication stub

The communication stub optimizes concurrent use of the network bus and abstracts its underlying technology. In this section, we describe how we connected our ECU to a CAN bus; Section V-C shows how the CAN connection can be replaced by a FlexRay connection.

The communication optimization focuses on minimizing the number of messages to be transmitted from dSpace to the ECU and vice versa. Messages that are to be sent at the same time are therefore piggybackked into one packet. The abstraction takes care of a uniform message format and it hides variations in latency and jitter involved with communication. This is needed, because the data structures, that define the messages being communicated over the CAN network, are compiled differently by the dSpace and the Freescale compilers. The communication stub therefore encodes and decodes CAN messages.

Moreover, without any additional means, the clocks at the dSpace box and our ECU will not be synchronized, which may lead to jitter. Although the central controllers at the dSpace and the local controllers at the ECU may roughly run at the same speed, they will not be as tightly synchronized as they would be in case both run on the same dSpace box. This may have two consequences for a local controller:

1) When it runs ahead, it may expect an absent message;
2) When it runs behind, it may receive multiple messages.

Both problems are resolved by assuming that the local controller has a state-message semantics. That is, the last value that has been sent is returned and there is no synchronization between sender and receiver.

This way of communication may lead to conflicts with the LSC, because the LSC expects messages upon each event that requires a change of its internal state. We have therefore implemented a translation layer in the communication stub in order to support event messages (see Section V-A).

## IV. Application mapping

In this section, we firstly describe the mapping of applications to tasks and servers. Secondly, we describe the mapping of applications to messages on the fieldbus. Finally, we discuss the mapping of applications to memory.

### A. Servers and tasks

As suggested by Figure 2, the application settings of MicroC/OS-II and the integrated ECU software are together defined in a main file. This file includes declarations of tasks and servers, their priorities and the stack size of the start tasks, i.e., the task that creates the other tasks and that starts the real-time clock. The real-time clock operates at 4000Hz, which restricts the monitoring of the resource consumption to 10% of the execution of the most frequent control loop.

In total we define 3 servers, i.e., given in descending priority order: for the RAP, the active suspension and the LSC process. The RAP and the LSC are (arbitrarily) allocated 10% processor bandwidth each period of 10 ms. Based on our experiments, the processor budget of the server corresponding to the active suspension control is set to 80% of the processor bandwidth with a period of 2.5 ms.

The local 4PH suspension control consists of two control loops (for current control and for pressure control) which are running on the same server. For each of the control loops, a task is created and their priorities are assigned in a rate-monotonic manner. In order to reduce the number of context switches between these tasks, their execution is forced in a strictly alternating manner (using a release offset and semaphore protection), so that 1 execution of the 100Hz control loop is followed by 4 executions of the 400Hz control loop. Moreover, the offsets of the tasks are chosen such that the high-frequent task cannot be preempted due to the server's budget depletion.

### B. Fieldbus communication

In our setup, the applications on the ECU report their status to the CSC. The fieldbus (by default CAN) is therefore used by three different applications, i.e., from the ECU's sides:

- **Active-suspension control:** every 2.5 ms, it reports the current and voltage set points of the valves.
- **LSC:** every 10 ms, it reports state and error information;
- **RAP:** sends an *"I'm alive"* message every 10 ms.

The messages from the LSC and the RAP are piggybacked on the control messages, because these have the highest frequency.

In return, the CSC on the dSpace box replies to our ECU every 10 ms. The messages received by our ECU contain:

1) set points and estimated valve flows for the control loops;
2) state-change commands for the LSC;
3) state-change commands for the RAP.

### C. Memory management

A major challenge encountered was that the control application (generated from Simulink) did not fit into the non-paged memory of the processor, i.e., the application requires more than the 8KB directly accessible RAM. Additional RAM can be used by means of the so-called *banked memory model* which enables memory paging. By loading a page into the page window and making sure no other page is loaded into this window, 4KB additional RAM can be directly addressed.

The support for memory paging required us to change the functions involved in context switching, including the interrupt service routines (ISRs), because the stack needs to store the PPAGE register and a 24-bit function pointer (only a 16-bit pointer is stored in the non-paged case). Paging has only been implemented for code, not for data as this would have required additional effort which was unnecessary to solve our memory problems. For performance reasons, compiler directives (pragmas) were applied to ISRs in order to link them into non-banked memory.

## V. Discussion: reliable communication

### A. Joint event-triggered and time-triggered message handling

In our design, two types of message semantics have been integrated [14]: event-message and state-message semantics. For event-message semantics, a message is associated with an event that is processed upon receiving the message. Also, synchronization is needed between sender and receiver. For state-message semantics, the last value that has been sent is returned which represents the last known state of the sender. Since we cannot assume intermediate synchronization between the dSpace box and our ECU, we implemented a translation from event-message semantics to state-message semantics.

The 4PH suspension control loops are implemented using Matlab/Simulink. Matlab/Simulink implements time-triggered activations of control tasks and it polls for input data, corresponding to state-message semantics. The local 4PH suspension control will therefore automatically read the data of the latest received message, i.e., following the state-message semantics.

However, the LSC assumes event-message semantics, because the supervisory control is assumed to be activated upon a (relevant) state change in its environment. This requires a conversion from state-message semantics to event-message semantics in the communication stub. For this purpose, our communication stub provides dedicated send and receive primitives, which we briefly describe below.

*1) Sending messages:* When a *send* primitive is called from the communication stub interface, this will simply cause an update of local data within the communication stub corresponding to the send request. However, no message is being submitted at this time. Only periodically, messages are being packed and submitted to the CAN bus.

When multiple state changes happen for the local read data of supervisory control, it gives rise to multiple events to reflect those state changes. This may lead to overload situations, as discussed in the next subsection.

*2) Receiving messages:* When a *read* primitive is called from the communication stub, all messages (if any) will be retrieved from the message queue of the CAN driver, in the order of arrival. Only the latest received message will be taken into account for handling, because a state-message semantics is assumed. This is possible because only the local state of the LSC needs to be updated. This message may cause a state-change in the LSC, where only the last state matters.

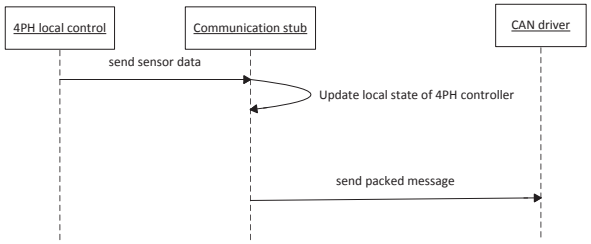


Figure 4. Interaction diagram for sending a CAN message.

Note that only a state change with respect to locally stored data (for example, by the LSC) is translated into an "*event*". In this way we effectively transformed state-message semantics into event-message semantics.

### B. Handling communication errors and overloads

Within our system, we cannot assume that message communication is reliable. When a task attempts to send a message to an uninitialized node in the network, a CAN error interrupt is generated. If this interrupt is not handled properly, this causes a crash of the control software. An ISR is therefore developed to handle this interrupt, i.e., it resets the CAN bus with a call to *CANStart*. The execution time of this ISR is considerable and it exceeds the execution time of the control loops. In practice this is not problematic, because uninitialized nodes typically occur only once when the applications are bootstrapped.

Furthermore, once the communication has been initialized, our ECU may be unable to keep up with the produced messages of the central control running on the dSpace box. In consumer-producer situations, a consumer (e.g., a LSC) may not be able to keep up with the producer (e.g., the CSC delivering commands over the network). A common technique to prevent buffer overloads is to selectively delete incoming messages. In this way, unacceptable latencies between the reception of the remaining commands and their handling can be avoided.

Deleting events in a state-message semantics is only possible, if and only if the new state of the receiver depends on just the latest event (rather than all intermediate states). Given that state changes of the LSC do not appear often, we experienced that in our proof of concept pruning of messages can be ignored.

### C. Replacement of CAN by FlexRay

FlexRay has been introduced in order to increase the available network bandwidth compared to CAN. The FlexRay technology defines a communication cycle which is divided into static and dynamic segments. The static segment enables time-triggered communication; the dynamic segment allows each node to transmit its messages in the remaining bandwidth using event-driven communications (like with CAN). In this work, we merely used the static segment. Freescale provides a library for this, which contains a set of functions and protocol-specific interrupt handlers to interact with the FlexRay controller.

The payload size of FlexRay slots is configured to be 16 bytes. This allows the resolution of the messages to be increased compared to CAN. Another advantage of this payload size is that encoding of the messages into the slots can be done efficiently. Consequently, all messages can be encoded and decoded by using the union of their data (i.e., fast piggybacking). The data structures of the FlexRay messages are defined as a union of a set of fields and a byte array. Such a union provides the possibility to approach the memory location at which the structure is stored as one of the fields or as a byte in the array. This makes the earlier described functions to encode and decode messages obsolete.

However, since FlexRay messages are larger than CAN messages, FlexRay communication requires more data memory compared to CAN. This reinforces the challenges related to efficient memory management of the applications running on our ECU (as discussed in Section IV-C).

## VI. Conclusions and Future work

In this paper, we revisited HSFs for facilitating timing predictable integration of automotive software components. Previously, we have published both the theoretical [15] and the practical impact [6, 7] of resource virtualization on the timeliness of synthetic components. In this paper we presented our experiences with employing our HSF in an ECU with real automotive software. A Jaguar XF carries our HSF with active-suspension software, which we captured on video. Future cars are expected to rely even more on timing predictable composition, not just for further vehicle dynamics but also for car-to-car control (like collision avoidance). Here real-time systems and the internet-of-things may join their forces.

## References

[1] M. Di Natale and A. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools," *Proc. of the IEEE*, vol. 98, no. 4, pp. 603–620, April 2010.

[2] T. Nolte, I. Shin, M. Behnam, and M. Sjodin, "A synchronization protocol for temporal isolation of software components in vehicular systems," *IEEE Trans. on Ind. Inf. (TII)*, vol. 5, no. 4, pp. 375–387, Nov. 2009.

[3] B. Bonsen, R. Mansvelders, and E. Vermeer, "Integrated vehicle dynamics control using state dependent riccati equations," in *AVEC*, Aug. 2010.

[4] dSPACE GmbH, "Automotive Solutions – Systems and Applications," 2015. [Online]. Available: https://www.dspace.com/

[5] Micrium, "RTOS and tools," 2011. [Online]. Available: http://micrium.com/

[6] M. Holenderski, R. J. Bril, and J. J. Lukkien, "An efficient hierarchical scheduling framework for the automotive domain," in *Real-Time Systems, Architecture, Scheduling, and Application*. InTech, 2012, pp. 67–94.

[7] M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Transparent synchronization protocols for compositional real-time systems," *IEEE Trans. on Industrial Informatics*, vol. 8, no. 2, pp. 322–336, May 2012.

[8] G. H. Broadfoot and P. J. Broadfoot, "Academia and industry meet: Some experiences of formal methods in practice," in *APSEC*, 2003, pp. 49–59.

[9] S. J. Prowell and J. H. Poore, "Foundations of sequence-based software specification," *IEEE Trans. on Software Engineering (TSE)*, vol. 29, no. 5, pp. 417–429, 2003.

[10] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, Int. Ser. in Computing Science, 1985.

[11] P. J. Hopcroft and G. H. Broadfoot, "Combining the box structure development method and CSP for software development," *ENTCS*, vol. 128, no. 6, pp. 127–144, May 2005.

[12] "Verum® - Tools for building mathematically verified software," 2009. [Online]. Available: www.verum.com

[13] "MISRA - The Motor Industry Software Reliability Association," 2004-2009. [Online]. Available: http://www.misra-c2.com/

[14] S. Poledna, "Optimizing interprocess communication for embedded real-time systems," in *RTSS*, Dec. 1996, pp. 311–320.

[15] M. M. H. P. van den Heuvel, "Composition and synchronization of real-time components upon one processor," Ph.D. dissertation, TU/e, The Netherlands, June 2013, ISBN 978-94-6108-443-9.